MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Memo No. 332 May 1975

IDEAS ABOUT MANAGEMENT OF LISP DATA BASES
(Revised version)

by

ERIK SANDEWALL

Abstract. The paper advocates the need for systems which support maintenance of LISP-type data bases, and describes an experimental system of this kind, called DABA. In this system, a description of the data base's structure is kept in the data base itself. A number of utility programs use the description for operations on the data base. The description must minimally include syntactic information reminiscent of data structure declarations in more conventional programming languages, and can be extended by the user.

Two reasons for such systems are seen: (1) As A.I. programs develop from toy domains using toy data bases, to more realistic exercies, the management of the knowledge base becomes non-trivial and requires program support. (2) A powerful way to organize LISP programs is to make them data-driven, whereby pieces of program are distributed throughout a data base. A data base management system facilitates the use of this programming style.

The paper describes and discusses the basic ideas in the DABA system as well as the technique of data-driven programs.

Work reported herein was conducted partly at Uppsala University, Sweden, with support from the Swedish Board of Technical Development, and partly at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

INDEX

1. Focus on the data base.

In this paper I will attempt to say three things at once. That stylistic experiment is undertaken not out of choice, but out of necessity: the three topics are intertwined, and none of them can be discussed without the context of the others.

The first topic regards the *attitude to data bases*: I shall argue that the current thinking about data bases in A.I. has missed an important point, which can be tersely characterized as the separate identity of the data base, independently of the program(s) that use it.

The second topic is a corollary of the first one, namely the design of systems for *management of data bases* in the new sense, in the context of a LISP or LISP-like programming system. A very experimental management system for LISP data bases is described. The system provides utility operations on the data base, such as data entry (prompt the user for contributions to the data base), presentation (nice printouts) and backup (dump a part of the data base on a file). Additional utilities are planned. All utilities use a description of the data base's structure, which is stored in the data base itself. The structure description must minimally contain syntactic information similar to what one finds in data-structure declarations in conventional programming languages. It can however be arbitrarily incremented by the user. Since it is in the data base, the description must itself have a description, which is also in the data base, and so on until a description which describes itself.

This system (called DABA) is motivated partly by the practical problem of maintaining collections of knowledge of non-trivial size, for use in A.I. programs, and partly by my

preference for a certain programming style, which is here called data-driven programming. Only a throw-away implementation of DABA exists currently; the system is described here in order to exemplify various desirable properties in systems for base management, and not as an available tool.

The method of *data-driven programming* is the third topic of the paper. That programming technique is frequently used but rarely discussed; the reader who has already used it will recognize it by a common operation in data-driven programs, namely

```
(APPLY (GET ...  ) ...)
```

In other words, data-driven programs are those where large parts of the program are procedures or program fragments that are stored in the data base, in a less trivial sense than as EXPR proprties. The paper argues for the use of this technique. This is relevant to the data base topic because program management tools for data-driven programs have the same requirements as data base management tools. In fact, the distinction between 'program' and 'data base' becomes fuzzy and unimportant.

The remainder of section 1 attempts to spell out my view of data bases, and the idea that utility programs are an important tool for working with a data base in the new sense. Section 2 describes the basic description mechanism in the DABA system; section 3 discusses data-driven programming in more detail; and section 4 discusses some simple procedure generation techniques in data-driven programs.

One of the many definitions of 'data base' in the world of commercial computing, is 'a collection of data which is suitable for use by a variety of different programs'. It is implicit in

the definition that the data base has an existence of its own, and a non-trivial life-length (although it may develop and change during its existence). The definition implies a need for separate documentation and separate maintenance of the data base.

This view of the data base is significantly different from what one finds in A.I. In our field, the 'data base' has usually been an appendix to or a scratchpad area for the program, created during the computation, and later garbage collected, or discarded at the end of the run. But the separate-identity view of the data base is appropriate also in the A.I. context, in the following cases:

- as the user-provided collections of knowledge that programs use. It has been common practice to use minimal knowledge bases when programs are run (for several reasons including memory problems), but the time now seems ripe for working with more exhaustive collections of knowledge. The problems of setting up, debugging, and editing the knowledge base then become non-trivial.
- as knowledge generated by or reorganized by programs. Learning programs (in the broad sense of the word) are only useful if the acquired knowledge can be saved for use during later runs. As another example, programmer's-apprentice-type programs [see e.g. Rich and Shrobe, 1974] need to analyze the user's input program, and form a model of it. That model has to be maintained between runs.
- as data-driven programs. Since programs have to be preserved between runs, it only makes sense to say that a program is a special case of a data base if the data base is so preserved.

Let the two kinds of data base be called a 'scratchpad' data base (temporary data base during execution of a program) and a 'perennial' data base (has separate identity, separate

documentation, etc., is maintained between runs, and is designed so that it can conveniently be used by several programs). In fact, the difference is as much in the way of looking at and working with the data base, as in the design of the data base itself.

The 'perennial' or 'separate-identity' view of a data base is very similar to the ordinary LISP programmer's attitude to his program. Working with a program does not merely involve running it, but also various types of service work: one may take out a part of the program and re-write it; one may take out a piece of another program, adapt it, and insert it in one's own; one uses pretty-print programs, cross-indexers and other tools, to obtain readable listings and documentation for careful study of the program, and so forth. The very same operations on a data base come naturally when it develops to non-trivial size.

The major computational implication of the 'separate-identity' view of the data base is therefore the usefulness of *utility programs*, i.e. programs like pretty-printers and cross-indexers, which serve the user when he works with the data base, and which are usually called directly by the user, rather than as subroutines. Utility programs for operations on LISP programs are in common use, and can sometimes be used for data bases as well (such as pretty-printers). But a number of additional utilities, as well as additional options in existing utilities, are useful for data base operations. The following are utility operations which I have often wished I had had, when working with LISP-type data bases, and which exist or are planned in the DABA system:

– a data entry utility that prompts the user for contributions to the data base. In a simple case, instead of letting the user type in

(DEFPROP BOSTON MASS INSTATE)

(in an elementary object-property representation), the system would acquire the information that BOSTON is a city, and then prompt the appropriate properties by typing out for example

BOSTON : INSTATE =

whereupon the user can answer

MASS

The difference in convenience and error rate is of course negligible for the extremely small toy bases that often have been used in A.I. programs, but significant when one enters more practical volumes of data. - In practice, a good data entry utility must allow for higher-level data representations as well, for mixed-initiative dialogue, and for conversational conveniences such as 'undoing' [Teitelman, 1974].

- a dumping utility for saving collections of data on files. If we again use an example in the elementary object-property representation, the filing utility needs a catalogue of carriers (such as BOSTON above) and information about which properties of this carrier shall be saved, and it should generate a file which when read will re-create those properties. A basic facility of this kind exists in INTERLISP [Teitelman, 1974].

- presentation utilities which print out the data base or parts of it in a nice format, so that the user can work with it easily. Several presentation methods are possible: an indentation-oriented layout is reasonable when one prints properties which are sizable expressions, and when when one wants to print properties of properties recursively to some depth. A tabular layout with several columns is appropriate for atomic properties, and for relation-type data bases where the data base is a set of tuples. Such presentation utilities are similar to the

dumper, except that they could also make use of information about the intended structure of properties. For example, if it is known in a separate declaration that the property under a certain indicator is to be a list which will be used as a set, then an appropriate indentation strategy could be chosen, and one might sugar the printout with curly brackets. If it is known that another property is a *gensym* atom, then one might want to print it in terms of some of its properties, rather than as its printname.

— a checking utility, to check that all properties in a collection of data satisfy the descriptions that have been made. One can check against declarations of the intended structure for each property (atom of certain type, list of atoms, etc.), against redundancy rules ("if A ∈ getp[B,I], then B ∈ getp[A,J]"), and so on.

— a merging utility. Suppose that travel cost between cities has been represented as

```
getp[BOSTON,TRAVELCOST]  =  [NYC [AIR 28.37  BUS 13.75]
                                TORONTO [AIR 189.10 ...] ...]
```

with the obvious interpretation (Boston - New York $ 28.37 by air, etc.), and that one wants to merge two files of data with similar structure. If both files contain properties for the same carrier/indicator pair such as BOSTON/TRAVELCOST, then one must make the obvious merge of the two assigned properties, rather than let one overwrite the other. A fairly general utility program could do that if provided with structure declarations for properties.

— an excerption utility. The inverse of merging (for obtaining a prescribed subset of the data base), but needs the same structure information.

== a utility for shift of representation. Suppose we want to re-represent the travel cost information above as

```
getp[BOSTON,FLIGHTCOST] = [NYC <US$ 28.37> TORONTO <US$ 109.10> ...]

getp[BOSTON,BUSCOST] = [NYC <US$ 13.75> ...]
```

either because of a whim when changing our own primary program, or in order to adapt somebody else's data to our program. Such a shift should again be doable by some utility, provided with descriptions of the old and new structure, and their relation.

The list can easily be continued. It is trivial to write programs for such operations, for each application or each data base one has. But it is a bother, and one would prefer to have access to more general utility programs. More general programs are slightly harder to write, since one wants them to be usable for various higher-level data representations besides the elementary object-property representation. Depending on the desired flexibility of the program, a utility program may range from a hacking exercise to a hard A.I. problem.

When a (general) utility program is used, it must be provided with a parameter-type description of the data structure that it is to operate on. That description can sometimes be integrated in the data itself, but often it is desirable to write it separately, like a set of declarations for the data representation. In the latter case, it is also possible to speed up execution by partially evaluating the utility program with respect to the parameters as described in [Beckman et al., 1974].

If one has to write out those declarations for each utility program, then that also can be a considerable burden. But it seems that the same declarations or structure descriptions could

serve several utilities. For example, in the elementary representation where properties are assigned to typed objects, one needs information about:

- which properties are carried by each type (used by data entry, dumping, and presentation utilities);
- which structure is expected for the property under a certain indicator (can be used by almost all utilities, including those for presentation, checking, merging, exception, and shift of representation. Also, it would be reasonable to check for appropriate structure during data entry).
- redundancy rules, for example for property inversion (used by the checker, as discussed above, and could also be checked or generated on data entry);
- if higher-level data representations are used, such as contexts, property assignments to non-atomic carriers, or relational storage with pattern-directed retrieval, then all utilities need to know about the storage conventions for that representation.

Furthermore, such a structure description for the data base is also part of the desired user documentation for the data base. It is therefore a reasonable goal to have one common description which can be used by all utilities, and for documentation purposes.

All points that have been made so far apply not only to LISP data bases, but also to conventional, 'bulk' data bases, and are in fact well recognized in the latter environment. The LISP environment does however offer some additional possibilities. Most importantly, the description of the data base can be stored in the data base itself, and still be used by the program that operates on the data base. To render this more precise, it is natural to consider the data base as a collection of *data blocks*, where the description of a data block is a new data

block which is also in the data base. (The regress terminates if some data block describes itself). The structure description of a data block will be called its *meta-block*. Utilities can then usually be defined as operations on blocks, which use the meta-block of the argument as parameters.

The idea of data blocks is in fact useful not only for distinguishing data from their description, but also for modularizing the 'primary' data (data which serve the purpose of the system, as opposed to descriptions) in the data base. A data block should then be a chunk of data which have a common structure and/or are closely related by some criterion. It could consist of a set of tuples (= relations) which are stored in the data base, or (in the elementary representation) of a set of property assignments (= triples of carrier, indicator, property).

A word of caution: the term 'block' has some connotations in computing which are not intended in this context. No recursive nesting of blocks or scope for identifiers is intended. It is in fact often desirable to distribute the properties of an atom to several blocks. The primary intended association of the term 'data block' is to the practice of organizing LISP function definitions into 'blocks' or 'files' of closely related functions.

An experimental system, called DABA, has been instrumental in developing and testing some of these ideas. DABA is a MACLISP program. The next section describes the data description and block structure in the DABA system, and also discusses other aspects which would be desirable in a more developed system. Before proceeding, let us however add some hand-waving comments about this whole approach.

From the theoretical viewpoint, the central issue in this work is *data base description*. Utility programs enter the picture because they probably offer the first practical usage of such descriptions, but there are also a longer-range aspects to data base descriptions in A.I. First, automatic-programming and programmers-assistant type systems need not only program descriptions, but also data base descriptions, whenever the program they generate or support is to operate on a data base - and for A.I. programs that is usually the case.

Another reason for working at data base descriptions is the need for methods of evaluating, comparing, and relating the many proposed representations of 'knowledge' data, such as various 'semantic nets'. Again it is natural to compare with the situation for programs: Active work on the theory of computation for almost a decade, has provided a considerable body of knowledge about equivalence, complexity, and other properties of programs. Similar knowledge about data structures would be quite useful. Just like for programs, mathematical logic can be expected to be of some use, but not to take us all the way. A concept of self-describing data bases might fit well into that picture.

The idea of defining operations (such as utilities) on blocks of data should perhaps be explored as a programming notation. Matrices (of numbers or other atomic entities) are a powerful notation in mathematics and in programming languages such as APL [Iverson, 1962]. Also, one of the advantages of relational data bases [Codd, 1970] is that they enable one to use an algebra on relations [= sets of tuples], which has also been used for some formal query languages. The same idea might be useful for specifying search and other quasi-parallell operations in LISP programs.

2. Servicing utility operations.

The DABA system can be used in at least two modes. In the simplest mode, the user has one program, here called the primary program, which uses the data base. A question-answering program is a standard example. As the data base attains non-trivial size, the user wants to use some utility programs on the data base. He therefore has to write down a structure description of the data base he already has. DABA is a system for representing and maintaining such descriptions in a systematic way, plus a collection of utility programs which use the descriptions. In the case discussed here, the primary program and the data base existed before the DABA facilities were called in. (The other mode of using the system is for managing data-driven programs, and will be discussed in the next section).

Let us choose a specific example and then describe how its structure would be described to the DABA system. We must here select a very simple example, which uses an object-property representation, in order to concentrate on the description. The DABA system is however useful for data bases with a richer structure as well.

Consider a block of property-list data about cities in the eastern United States. The block is a set of property assignments, or triples, such as

```
[<BOSTON, INSTATE, MASS>,
 <BOSTON, SUBURBS, (LEXINGTON, REVERE,... )>,
  ...
 <NYC, INSTATE, NY>,
  ...
 <MASS, HASCITIES, (BOSTON, LEXINGTON, ...)>,
 <MASS, FULLNAME, MASSACHUSETTS>,
  ...  ]
```

which of course says that Boston is in the state of Massachusetts, and so on. ('...' indicates

continuation and is not intended to be in the data base). Each data block has a *name*, which may be atomic (but does not have to be). Let the atom US-EAST be the name of the above block.

A QLISP-like notation will be used, with angle brackets <...> for tuples = lists, curly brackets {...} for sets, and square brackets [...] for free property-lists. A property-list [i1 v1 i2 v2 ...] is a set of assignments of vk to ik, so the square bracket expression is really an abbreviation for

{<i1 v1>,<i2 v2>,... }

LISP function definitions will be written with round parentheses (...). All these types of parentheses are assumed to map into ordinary parentheses in the actual implementation. In other words, the knowledge that a certain list represents a set rather than a tuple, is not assumed to be available in that item itself.

It will be more convenient to specify the contents of blocks using the access function dgetp[c, i, n], where c is a carrier, i an indicator, n a block name, and the function returns the corresponding property-value. The block contents above can therefore be described as

```
dgetp[BOSTON, INSTATE, US-EAST]   =  MASS
dgetp[BOSTON, SUBURBS, US-EAST]   =  (LEXINGTON, REVERE, ...)
 ...
```

The description of a block in DABA consists of two parts. Consider a data block (of which US-EAST is a toy example) and a program which uses the block as a data base for question answering or some similar purpose. One could write down several different blocks, using the same conventions, and the program would then presumably be able to use any of these blocks. The *description of representation* shall contain a specification which is common to these blocks, and which therefore encodes some of the conventions that are assumed by the program. By

contrast, the *description of extent* contains a catalogue of the contents of each block, and other information which is local to the block. There are several reasons for making such a distinction: economy of storage for the shared part of the description is an obvious reason. Also, the previously mentioned possibility of partially evaluating a utility or other parameter-driven program with respect to the data base description, is only worthwhile if the part of the description that is being kept fixed, can be factored out. (There are however also ways of avoiding the distinction, in special cases when one does not want to make it).

The common denominator for the two descriptions is the *sorts*. In the present example, one immediately recognizes different sorts of carriers: CITY, STATE, etc. The description of extent for a block includes a catalogue of the carriers in each of the sorts, represented as:

```
ngetp[US-EAST,NODES] ←
    [CITY (BOSTON, NYC,...), STATE (MASS, NY,...),...]
```

while the description of structure includes the information of what indicators are used by objects in each sort, for example that objects of type CITY may carry properties under the indicators INSTATE, SUBURBS, etc.

The function ngetp is used for getting properties of blocknames, in the description of the blocks extent. The function may sometimes simply make an access in the property-list of its first argument, in which case it is synonymous to the INTERLISP getp, but it may also compute its value by default from an appropriately stored procedure, handle non-atomic block names, etc.

The description of extent also includes information about the location of the block, for example 'as global property-lists', 'as property-lists local to this block', or 'as text file with name ...'. The

first case is expressed as

```
ngetp[US-EAST,ATLOC] = GLOBAL
```

The conventions used in the description of extent are to some degree arbitrary. One might prefer to split up the NODES property so that the set of sorts is obtained in one access, and the set of carriers in a sort is obtained in one access for each sort. Such changes would not be significant.

The meta-block of US-EAST (= its description of representation) is another block, whose name might be CITIES. The relationship is indicated by

```
getp[US-EAST,META] = CITIES
```

Some minimally needed information in the meta-block is, first, which indicators are carried by objects in each sort in the described block. Thus, since BOSTON and NYC have properties under the indicators INSTATE and SUBURBS, and since they are in the sort CITY, one should have:

```
dgetp[CITY,CARRPROPS,CITIES] = {INSTATE,SUBURBS,...}
```

and likewise

```
dgetp[STATE,CARRPROPS,CITIES] = {HASCITIES,HASCAPITAL,...}
```

and so on.

The meta-block should also contain information about the expected structure of properties. In our example, we know that properties under the indicator INSTATE shall be atoms of the sort STATE, that SUBURBS properties shall be sets of cities, and so on. Such conventions could be encoded in a straight-forward fashion as

```
dgetp[INSTATE,PROPSTRUC,CITIES] = <SORT STATE>
dgetp[SUBURBS,PROPSTRUC,CITIES] = <SET <SORT CITY>>
```

In our simple example, all names (block names, carriers, indicators, sort names, etc.) have been atoms. That is however not necessary, and in descriptions of less trivial representations it is frequently useful to let them be non-atomic.

The meta-block contains information which might occur as declarations in some other programming languages, and in the data description language of a management system for large data bases. The important difference is that here the meta-description is a new data block, so that the user can use and extend that information according to his own needs. For example, it would be natural to extend the meta-block with information which relates the primitives for this data block (sorts and indicators, in this simple example) to user-oriented concepts in a model of the intended application.

In the actual system, each block may be associated with a number of 'satellite' blocks which provide additional but optional information. User additions to a meta-block are usually best organized as a new satellite block, rather than as a change in the original meta-block. Even the PROPSTRUC property is actually kept in such a satellite block.

Very often one wants to define access procedures for properties, which compute the property from other data in the system, looks up default values, stores properties in alternative locations, etcetera. The meta-block therefore always contains an *access function* for each indicator, for example as:

```
dgetp[INSTATE,ACCESSFN,CITIES] ← XGETP
```

where xgetp is the default access function which does a trivial (eXplicit) look-up. Suppose

however that one would want to define a block US-EAST2 as an update of US-EAST, so that properties in US-EAST2 use properties in US-EAST as default. The block US-EAST2 would be described similarly to US-EAST, with the following amendments:

(1) ngetp[US-EAST2,MODIFOF] = US-EAST. This property assignment belongs to the description of extent of US-EAST2.

(2) getp[US-EAST2,META] = CITMOD. US-EAST2 needs a different description of structure. (In practice, its meta-block would have a non-atomic name, but we assume an atomic name here for simplicity).

(3) dgetp[INSTATE,ACCESSFN,CITMOD] =

```
(LAMBDA (C I N)(OR (XGETP C I N)
        (DGETP C I (NGETP N 'MODIFOF)) ))
```

and similarly for every other indicator that was assigned an access function in the old meta-block CITIES. This access function takes the same arguments as the function dgetp. It first checks if the property exists explicitly in the block that is mentioned as third argument, and otherwise looks it up in the default block. (In the actual system, access functions have a fourth argument, and can be used fo 'get', 'put', 'delete', and 'change' operations).

The block CITIES, which is the meta-block of US-EAST, should also in its turn have a meta-block and a catalogue (description of extent). The sorts in the block CITIES are SORT (containing the carriers CITY, STATE, etc.) and INDICATOR (containing the carriers INSTATE, SUBURBS, HASCITIES, etc). This structure is correctly described if we have

```
getp[CITIES,META] = OMEGA
dgetp[SORT,CARRPROPS,OMEGA] = (CARRPROPS)
dgetp[INDICATOR,CARRPROPS,OMEGA] = (ACCESSFN PROPSTRUC)
```

plus the appropriate properties on ACCESSFN and PROPSTRUC. It is then correct to define

```
getp[OMEGA,META] = OMEGA
```

so that OMEGA describes itself. In general, proceeding from a blocks to their meta-blocks, one always eventually reaches OMEGA, but often the path is longer than in this example. - The definition of the NODES properties for CITIES and OMEGA are straightforward.

What has been described so far is a basic description system, which might be sufficient for data blocks that use simple representations. In an environment where the user has already designed his primary program and his data base, he has to set up the description of representation as a *post factum* description of the conventions he has made. If he needs non-trivial access functions, he has to write them himself, although with skill and luck he may be able to define them as small interface procedures that call appropriate parts of his primary program. Similarly, the NODES property (= the catalogue) in the description of extent can sometimes be computed when needed, from information that has already been set up by or for the primary program, and otherwise the user has to create it.

Such a basic description is what is needed by utility programs as discussed earlier. The intended purpose of the DABA system is partly to provide a coordinated set of such utility programs, and partly to provide 'canned' higher-level descriptions. For example, in specifying the block CITMOD in the last example above, the user should only has to specify that it modifies CITIES (expressed by an appropriate property assignment to the atom CITMOD), and that the meta-block of CITMOD is e.g. MODIF, where MODIF would be a meta-meta-block which imposes the appropriate defaults for access functions, NODES properties, etc. in CITMOD. Similar meta-meta-blocks are or should be available for other common operations inside and

between data blocks.

3. Program/data base integration.

The DABA system is not particularly helpful for developing conventional programs. It is however believed to be useful when one uses an often used, but little recognized programming technique that I call data-driven programming. In this section I argue that data-driven programming is a significant development, and much more than a hack; and also that a DABA-type system can facilitate the use of this method.

A common model for a program in LISP (and most other languages) is that the program is a set of procedures which call each other. Each procedure has a name. A call from a procedure FOO to a procedure FIE is manifested in that the definition of FOO explicitly mentions the name 'FIE'. Such a textbook model of programs is not always applicable. Many programs are organized as a collection of procedures each of which is attached to data items in a data base, plus perhaps one part which is an ordinary program. In such a program, a procedure f may sometimes process its input data by calling procedures which are attached to them in the data base. This constitutes an indirect or data-driven call from the procedure f to a procedure g. Usually the procedures or program fragments are stored as properties of atoms, but they may appear anywhere in the data base.

A *data-driven program* then consists of some 'ordinary' procedures, and some 'data-driven' procedures which are invoked through data-driven calls. In most programming languages it is difficult or impossible to implement data-driven programs, except of the very restricted kind that are obtained in *case* statements where the driving data are integers (Fortran, Algol 68) or a set of items that have been explicitly declared in the program (Pascal). It is easy and

straightforward to implement data-driven programs with full generality in interpreted LISP, but this programming practice is not fully recognized: INTERLISP's *makefile* system [Teitelman, 1974] provides a lot of service in keeping track of compiled code, but assumes that it is stored in the 'function cell' of the atom. In MACLISP [Moon, 1974] the compiler has only very recently been provided with an option that allows it to compile functions that are not EXPR or FEXPR properties. One should not treat data-driven programming as 'hack', thereby implying that it should be discouraged, or that it lacks research interest. It is a powerful programming method and program structuring method for the following reasons:

--- *Procedures obtain truly meaningful names.* In data-driven programs, each procedure is identified not by a single name, but by a combination of such. For example, procedures that are stored directly on property-lists are identified by pairs of atoms. Therefore, the identifier of a procedure can be more than 'mnemonic': it can state the purpose of the procedure in a fashion which can be used by other parts of the program.

For example, McDonald's bibliography program [McDonald, 1975] assumes that each bibliography entry is associated with a number of properties such as AUTHOR, TITLE, etc., and each property name has on *its* property-list procedures for reading, printing, etc. that property. The procedure that is identified as ge[AUTHOR,PRINT-UP-FN] has such a better-than-mnemonic name. The routine which goes through all desired properties of an item and applies the reading procedures of each indicator, uses the meaning embodied in the 'name'.

--- *Facilitates automatic program generation.* If program generation is to go beyond the level of toy programs such as trivial sort routines, the generator must use a model of the program that is

being generated. The task of specifying the model, and even more the task of relating the model to the program, are particularly simple for data-driven programs. The actual program generation can then often consist of generating individual data-driven procedures or code fragments. The latter case arises if each data-driven procedure has the form

(LAMBDA (X ..)(FOO (code 1) (code 2) ... (code n)))

where each expression (code i) has been generated separately, and where the function FOO is the 'glue' between the programs and is responsible for communication between them. (FOO may be a built-in function such as OR or PROGN, or a function written for the purpose). The PCDB system [Sandewall 1971, Sandewall 1973, Haraldson 1974] uses this method for program generation.

— *Uses the application language, and makes it easily extensible.* The notation that is input to a program is or should be a language which is natural to the application of the program. The same holds for the notational conventions that are used in a data base. In both cases, a program which is organized around such an application-oriented notation is likely to have a good structure, and extensions to the program immediately reflect extensions to the application 'language'.

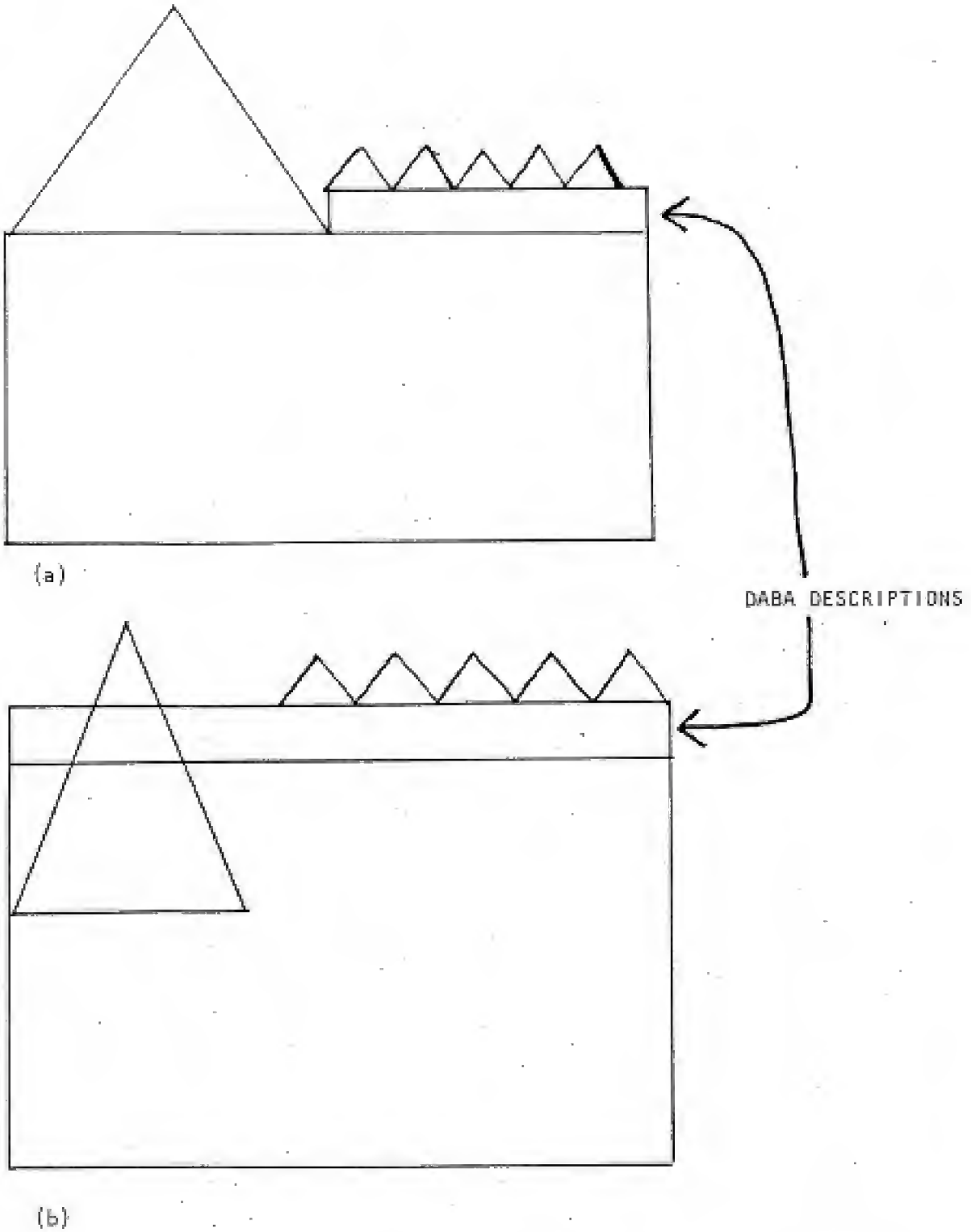
Interpreters are a classical example of data-driven programs. Interpreters for conventional languages are data-driven with respect to procedure names (i.e. data of the interpreter). Recent language features such as pattern-directed invocation and demons also assume that procedures are indexed from data structures, although in this case the data of the interpreted program. The apparent power of the latest generation of A.I. languages [Bobrow and Raphael, 1973] is perhaps largely due to the fact that they made data-driven programming available to users who

did not think of using it explicitly. The claim here is that it is often better for the user to develop his own scheme for organizing his program (in the sense of storing the procedures in the right places) instead of using a single package of high-level devices. There are also several examples of successful data-driven programming around. The SHRDLU program [Winograd, 1972] can be used in support of many claims; it is also data-driven in several parts.

The reason why this whole discussion is brought up in a paper about data base description, is that data-driven programs allow procedures to appear in arbitrary positions in the data base. There are often plenty of relationships between procedure items and other items in the data base: procedures may have been generated from other data, and program analysis programs may often generate data about programs that should be stored in the data base, so that it does not have to be re-generated repeatedly. It is then natural to use one's data base management system for managing programs and program descriptions as well.

The contrast between the situation described here, and the situation described in the previous section, is characterized by figure 1. In diagram (a), the large triangle is the primary program, the small triangle the utility programs, and the data base is described by the DABA system for the utility programs. In diagram (b), the primary program consists mainly of data-driven procedures which are also managed by DABA.

Figure 1



There is also another reason: the data that data-driven procedures are associated with, can sometimes be 'object' data for the system, but very often it is natural to choose them as items that appear in the self-description of the data block, for example indicators or sort names. Thus the descriptions of a data block are often an appropriate framework for organizing the program.

Most utility programs can with advantage be data-driven. For example, a presentation utility could be driven by printing procedures associated with property indicators. This is a commonplace idea, but raises some practical problems. Consider the following scenario: we have acquired a large data base (large by A.I. standards, that is), consisting of several blocks with different structures. We are also using a number of different utilities, each of which drives specialized procedures for all or some of the blocks. Furthermore, descriptions of the data blocks sit around and are directly interpreted by several of the utilities, and are used for generating specialized procedures for some others. Suppose now that we want to move this battleship a bit, for example: (a) modify the structure of some data block, (b) delete a data block, (c) discard a utility. The first operation implies a number of other changes in the system; the other two enable non-trivial garbage collections. In a large system with a considerable life-length, such garbage collections are necessary (even if one has infinite memory, he still wants to know what is garbage so it does not have to be updated).

In order to support such such simple operations, and also in order to support the user who wants to understand the system, so that he can perform more complex operations on it, one needs a model of the structure of the system. Here again the block structure and other concepts in DABA are useful.

Let us exemplify that, again with a simple example. Consider a pretty-printing utility program P, which operates on a data block B whose meta-block getp(B,META) = M. The program P makes use of specialized printing procedures and other parameters which apply to all blocks which like B have the structure described in M. These parameters together constitute a data block MP. (They might be included in M itself, but it is not desirable to clobber M with auxiliary procedures for all utilities, and therefore we prefer to let each utility define its own 'satellite' block to M).

The block MP has the same sorts and the same catalogue as M, but uses different CARRPROPS assignments. For example, in our initial geography example, the block M contained PROPSTRUC and ACCESSFN properties, for HASCITIES and other indicators used in B. The block MP would contain a PRINTFN property for HASCITIES, which P then uses. The relationship between MP and M should be expressed by a reference such as

```
ngetp(MP,DESCRIBES) = M
```

This reference should imply a default value for the NODES property of MP.

The *meta-block* for MP must be a block which describes the structure of the parameters that the program P assumes, i.e. it is part of the documentation of P. In the present DABA system, utility programs are integrated with their specification, so a data-block P contains both the set of procedures that make up the utility program, and the information that makes P a suitable meta-block for MP. For example, P contains a reference to the knowledge about how to compute the NODES property of MP from its DESCRIBES property. (Actually, that knowledge is conveyed to P by *its* meta-block). The structure of these blocks is illustrated in figure 2.

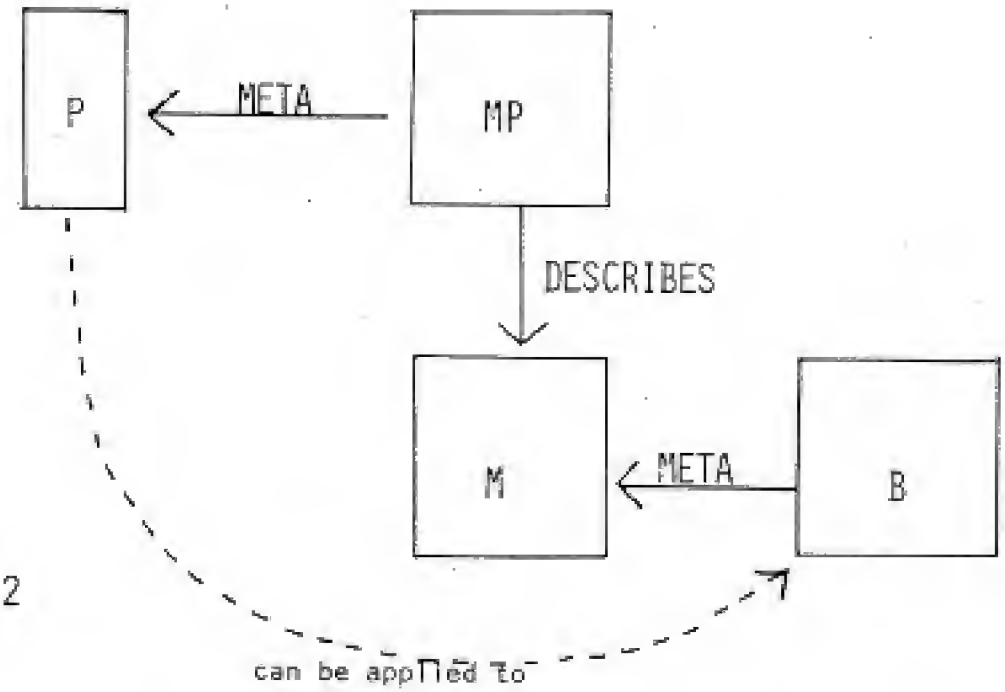


FIGURE 2

This example illustrates how blocks of data are not merely clusters with dense internal connections: there are also relationships beween blocks, such as the META relation, the DESCRIBES relation, the MODIFOF relation (used in an example in the previous section). Several other relations are important, such as the relations between a program, the block of data that was input to it, and the block of data that it produced as result. Relations between blocks are macro-level descriptions, which complement the micro-level, declaration-type descriptions such as CARAPROPS or PROPSTRUC properties.

4. Generation of procedures.

The DABA system as such assumes that data base descriptions contains procedures, namely access functions, and specialized procedures for various utility programs. In addition, many applications may involve data-driven programs as discussed in the previous section.

Where do these procedures come from? The simplest case is of course where they are always written by the user. There are however several ways whereby the user can be relieved of this responsibility, or at least of some of the drudgery involved.

One obvious method is by default computation. If the procedure does not exist, then it is computed by a procedure which may derive it from other data, ask the user, etc. This is accomplished in a simple and uniform fashion in DABA through a *recursive access-function mechanism*. The function dgetp which was used in section 2 to obtain data from the block USCITIES, is defined approximately as

```
dgetp[c,i,n]  =
   if n=OMEGA then getp[c,i]
   else  apply[ dgetp[i,ACCESSFN,getp[n,META]],  list[c,i,n]]
```

In other words, in order to dgetp the HASCITIES property of MASS, one retrieves and uses the ACCESSFN property of HASCITIES in the meta-block. But for that, he must retrieve the ACCESSFN property of ACCESSFN in the meta-meta-block, and so on. (At least theoretically; the recursion is sometimes shortcut). The recursion terminates at the ultimately 'meta' block OMEGA.

This mechanism is a flexible way of defining appropriate access functions. For example, in

section 2 we discussed the modified data block US-EAST2, which modified the block US-EAST, and where

```
getp[US-EAST,META] = CITIES
getp[US-EAST2,META] = CITMOD
ngetp[US-EAST2,MODIFOF] = US-EAST
```

Here the user should not have to write out the access functions for CITMOD. Instead, there should be a data block MOD which describes modification blocks in general, so that getp[CITMOD,META] = MOD. The access functions in CITMOD are obtained as dgetp[ACCESSFN,ACCESSFN,MOD], and might be the one outlined in section 2, or (improved) the following: go and get the access function for the same indicator in the block CITIES. Try using it in the current block (in this case, US-EAST2). If no result, then make an access in dgetp[current block, MODIFOF].

In fact, all other system properties, for example CARRPROPS, are accessed in the same way using dgetp. It is therefore not necessary to invent a new atom as a name for CITMOD. Its name is chosen as (MOD CITIES), whereby it is implicitly specified to be a block whose meta is MOD and which modifies the block CITIES. (The actual DABA notation is slightly different). In general, the method of defining properties of blocks through access functions in the meta block, complements well the method of using non-atomic ('molecular') names for blocks, where the contents of the block, or at least some of the contents, are implicit in the name of the block. The advantages with non-atomic block names are analogous to the naming advantages of data-driven programs.

Utility programs which use specialized parametric procedures also access them with the function dgetp, which means that the same kinds of default mechanisms can be used for their

parametric procedures such as PRINTFN. The recursive access mechanism is quite powerful, and enables one to implement a number of desirable facilities with a very small kernel system. Its major drawback is that higher-order access functions of access functions are usually less than transparent to read and understand. Efficiency may also be a problem, which hopefully can be handled by saving access functions so they only have to be computed once ('memoization'), and using automatic simplification of the lower levels of access functions.

Sometimes a procedure is to be built up and modified in several successive steps. It must then be initialized in some way (for example by its meta-level access function), whereupon it can receive *advise* which successively modifies it. For example, if a data-driven procedure contains or refers to a set of theorems or demons that are to be triggered by the indexing data, then each advise might contribute one more theorem or demon to the structure. A program for simplifying LISP expressions might associate with each LISP function a simplifier procedure for forms where it is the leading function. A new simplification rule, such as

```
(CAR (LIST $X $$Y))  ->  $X
```

would then be sent as a message to the simplifier for CAR. (The REDFUN program [Sandewall 1971, Beckman et al. 1974] works in this fashion). Sometimes the advise that is given to a procedure is less uniform. INTERLISP [Teitelman 1974] contains facilities for user-specified advise to the entry and exit parts of arbitrary user procedures. In the DABA system, it is frequently desirable to let various items send advise to an access function or class of access functions, telling it where to find explicit and default data, whether and how to 'memoize' computed data, and so on.

Several of Hewitt's *actor* ideas [Greif and Hewitt, 1974] carry over to this purpose. What we

have called advise is a kind of message. Giving advise is like an actor 'handshake': the receiver of the messages must be the one who knows how to incorporate it into his internal structure. There is a need for actors in the sense of objects which both receive and send messages. But chains of messages which trigger each other are here only a secondary purpose; the primary purpose of a message is to modify a procedure or other data item. Also, it is mandatory in our case to have an option for saving a protocol of which messages were sent where, so that later changes early in a message chain can perpetuate along the chain. Such a protocol should of course be stored as another data block, in line with the general philosophy of the system.

In fact, the data structure becomes cleanlier if messages are sent to blocks, rather than to individual procedures. Thus a simple meta-block might receive messages saying 'tell all your access functions to pick default values from the block B' or 'tell your access function for SUBURBS to get the explicit value, and filter away from it all proposed suburbs which are not in the same state'. The structure of the blocks involved should be as follows:

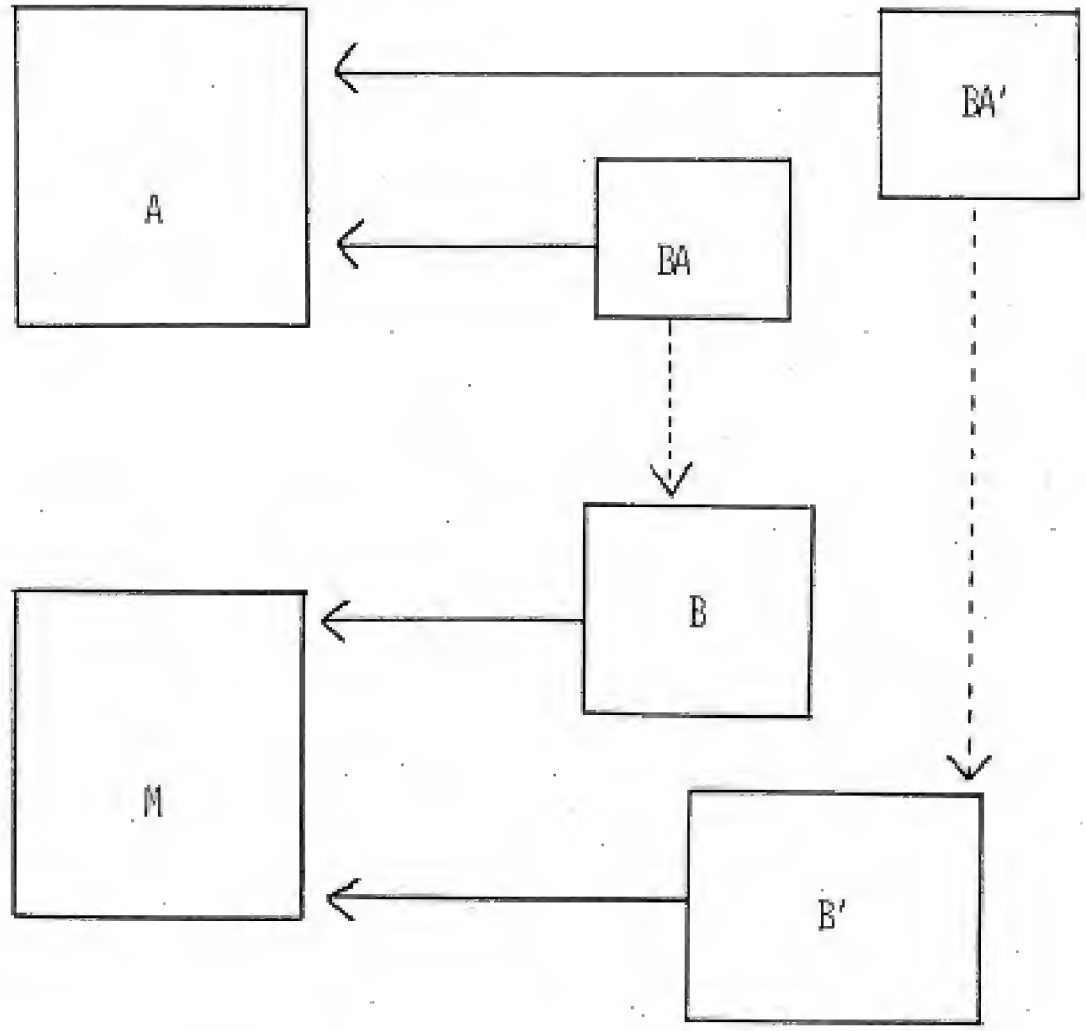
B block to which messages are sent

M getp[B,META]

BA block of messages that have been sent to B

A getp[BA,META]

Here A should contain the information about how to interpret incoming messages, and it should be on the same level as M. In other words, if B and B' have the same M, and BA' is like BA but for B', then BA and BA' should be able to share the same meta-block A (figure 3).



POINTS FROM BLOCK TO ITS META-BLOCK.

POINTS FROM A MESSAGE-HISTORY BLOCK TO THE BLOCK THAT SENT AND RECEIVED THE MESSAGES.

FIGURE 3

The previously mentioned methods for defining blocks through their meta-blocks, sometimes using non-atomic block names, are useful for protocol blocks as well. Protocol blocks are actually given non-atomic names such as (PROTOCOL B) rather than BA. The META of this block is then implicitly PROTOCOL. If M has an idiosyncratic structure, then a corresponding, tailored meta-block for protocols will be needed, i.e. PROTOCOL is really (PROTOCOL* M), and what we used to call BA has a name of the form ((PROTOCOL* M) B). The meta-meta-block PROTOCOL* computes parts of the protocol meta-block (PROTOCOL* M) from M.

Protocol meta-blocks such as PROTOCOL or (PROTOCOL* M) also contain the decoders for incoming messages. Each transmission of a message is called an *event*. An event knows what message it conveys, its source, and its destination (where both of the latter are block names). The event is also a member of the protocol blocks of its source and its destination, although with different properties in those two. A simple executive keeps track of a queue of events, and for each event looks up the decoder in the protocol block of the destination block of that event, and applies it.

This message-sending facility is *not* intended as some kind of programming system. If the DABA system is used as in section 2 of this paper, then it does not affect the user's primary program at all. It is intended as a mechanism for performing and keeping track of updates to the data base (including data-driven procedures), so that later changes in the data base (in the separate-identity sense of the word) obtain appropriate secondary effects. Also, messages are only sent 'to procedures' (loosely speaking) for changing them, not for invoking them.

5. Other aspects.

Some aspects of the DABA system have been more or less ignored in this paper. We have remarked that each block needs a description of structure, and a description of extent. The description of structure is the meta-block, and has *its* meta-block, and so on. The description of extent, or 'catalogue', is at least in simple cases the set of properties of the blockname. But it also needs a meta-block, where for example the access function for the name's NODES property is located (in the cases where the NODES property is computed from other information). The meta-block of a block B, and the meta-block of the catalogue of B are not in general identical, but the latter is derivable from the former. Also, the catalogue block of the catalogue block is computed as needed (storing it explicitly would lead to an infinite regress). The resulting structure is powerful, but unfortunately also tends to become fairly complex. Later generations of the system will attempt to simplify it.

Another aspect which has not been covered is the relationship between the description structure of DABA on one hand, and problems in the representations of knowledge, such as IS-A link problems and frame systems [Minsky 1974] on the other hand. Information in DABA meta-blocks such as CARRPROPS and ACCESSFN information corresponds vaguely to what one needs in those cases, but the correspondance is not trivial.

DABA is presently a MACLISP program, although it should be relatively easy to transfer it to other LISP dialects. It contains simple utilities for data entry, checking, dumping, and presentation. The utilities are data-driven and their structure is described within the system, as described above. The current system also contains facilities for keeping track of all blocks; and

a few general-purpose facilities such as comment blocks (for arbitrary other blocks) and update blocks. The message-sending facility for updates of procedures has been specified and is probably the next to be implemented. After that, the present implementation will probably have served its purpose, and the next generation of DABA will be due.

Acknowledgements.

Several members of DLU in Uppsala and the MIT A.I. group have taken the time to experiment with the DABA system, discuss the issues raised in this paper, and look over the manuscript. Special thanks are due to Dave McDonald, Charles Rich, and Gerry Sussman of MIT, and to Anders Haraldson and Jaak Urmi of DLU.

References.

Beckman, L. et al. A partial evaluator, and its use as a programming tool. Datalogilaboratoriet, Uppsala university, memo 74/34.

Bobrow, D. and Raphael, B. New Programming Languages for Artificial Intelligence. Computer Surveys, Vol. 6, No. 3 (September, 1974).

Codd, E. A relational model of data for large shared data banks. Comm. ACM *13*, No. 6, June 1970, pp. 377-87.

Haraldson, A. PCDB - a procedure generator for a predicate calculus data base. IFIP 74 proceedings, pp. 575-79.

Greif, I. and Hewitt, C. Actor semantics of PLANNER -73. MIT A.I. Lab, working paper 81 (November, 1974).

Iverson, K. A programming language. Wiley, 1962.

Minsky, M. A framework for representing knowledge. MIT A.I. Lab, memo no. 306 (June 1974)

McDonald, D. Private communication.

Moon, D. MACLISP reference manual. MIT Project MAC, April 1974

Rich, Ch. and Shrobe, H. Understanding LISP programs: towards a programming apprentice. MIT A.I. Lab, working paper 82 (December 1974)

Sandewall, E. A programming tool for management of a predicate-calculus-oriented data base. IJCAI 1971 proceedings.

Sandewall, E. Conversion of predicate-calculus axioms, viewed as non-deterministic programs, to corresponding deterministic programs. IJCAI 1973 proceedings.

Teitelman, W. INTERLISP reference manual. Xerox Palo Alto Research Center, 1974.

Winograd, T. Understanding natural language. Academic Press, 1972.